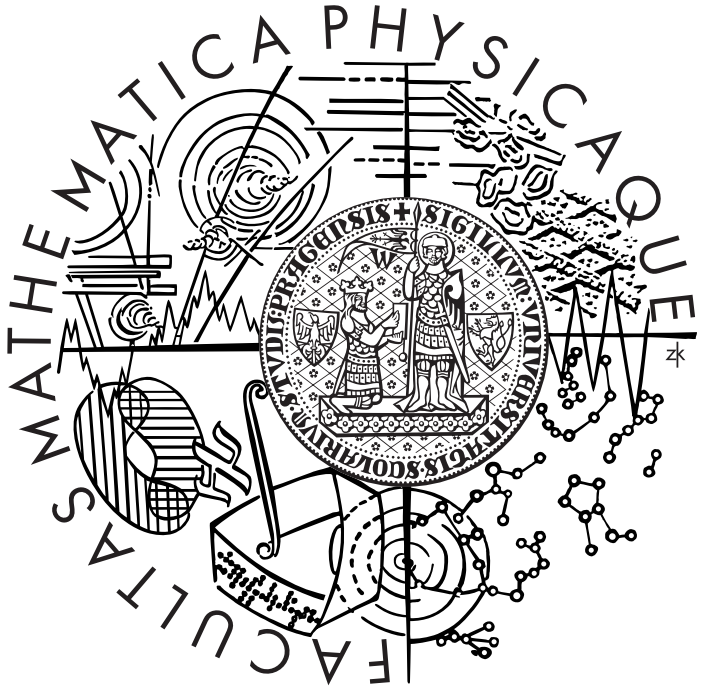
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Matej Pivoluska

Visual Development of Hierarchical Components

Department of Software Engineering
Supervisor: RNDr. Petr Hnětynka, Ph.D.
Study Program: Computer Science

I would like to thank Petr Hnětynka, Michal Malohlava, and Tomáš Bureš for all their suggestions, Miro Hudák for icons, Radovan Šesták for figures, and Matúš Maciak for all errors that he found. And I wish to thank my friends and family for all their support. Without them, this thesis could not be finished.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 18. 4. 2008 Matej Pivoluska

Název práce: Visual Development of Hierarchical Components

Autor: Matej Pivoluska

Katedra: Katedra Softwarového Inženýřství

Vedoucí diplomové práce: RNDr. Petr Hnětynka, Ph.D.

e-mail vedoucího: hnetynka@dsrg.ms.mff.cuni.cz

Abstrakt: Komponentové systémy umožňují vývoj aplikací pomocí sestavování existujících komponent do funkčních celků splňujících požadavky na danou aplikaci. Univerzální komponenty je možné opakovaně využít na vývoj různých aplikací, které vedou ke zkrácení jejich vývoje i k lepší kvalitě výsledků. Pro vývoj komponent a aplikací z nich složených je nutná dobrá podpora uživatelských nástrojů. Cílem této práce je navrhnout a implementovat prototyp IDE nástroje pro distribuovaný komponentový systém SOFA 2.0 (SOFA IDE). SOFA IDE poskytuje možnost vyvíjet aplikace vytvořené složením SOFA 2.0 komponent v grafickém uživatelském prostředí a správu komponentového uložiště. Implementaci předchází analýza komponentového systému SOFA 2.0, uživatelských požadavků a analýza možných podpůrných technologií pro implementaci. Prototypová implementace SOFA IDE se skládá ze třech integrovaných částí: (1) Prohlížeč komponentového uložiště; (2) Grafické uživatelské rozhraní pro jednoduchou tvorbu a verzování komponent a jejich součástí; (3) Editor diagramů architektury SOFA 2.0 komponent umožňujících vizuální tvorbu hierarchických komponent. SOFA IDE je implementované jako plug-in do Eclipse. Editor diagramů architektur je založený na projektu Eclipse GMF.

Klíčová slova: komponenty, distribuovaný systém, IDE

Title: Visual Development of Hierarchical Components

Author: Matej Pivoluska

Department: Department of Software Engineering

Supervisor: RNDr. Petr Hnětynka, Ph.D.

Supervisor's e-mail address: hnetynka@dsrg.ms.mff.cuni.cz

Abstract: Componenent systems allow development of new applications by assembling existing components together to form a desired result. General purpose components can be reused for development of different applications which leads to shorter development time as well as higher quality of developed applications. Good support built-in into user's tools is required for this process. The goal of this thesis is to design and to implement a prototype implementation of IDE tool for SOFA 2.0 distributed hierarchical component system (SOFA IDE). SOFA IDE allows visual development of applications composed of hierarchical components and managing SOFA 2.0 component repository. The implementation is preceded by analysis of SOFA 2.0 component system, user requirements analysis and analysis of possible supportive technologies for the implementation. The prototype implementation of SOFA IDE consists of three integrated parts: (1) SOFA 2.0 Repositories Browser which allows user to manage a content of connected SOFA 2.0 repositories; (2) graphical user interface for easy creating and versioning of components and their elements; (3) Architecture Diagram Editor for visual development of hierarchical components. SOFA IDE tool is implemented as a Eclipse plug-in. The Architecture Diagram Editor part is based on Eclipse GMF project.

Keywords: components, distributed systems, IDE

Contents

1 Introduction

Software systems (especially distributed ones), having a lack of physical constraints of their counterpart in a material world, are ones of the most complex construction tasks created by humans. A creator of a system has to describe the function of it, and to clearly state its interaction points and the way how to use it.

These days, component-based development is proved to be usable for building of large software systems [25]. The idea of software components as reusable software building blocks organized in component libraries is attributed to Doug McIlroy and his talk Mass Produced Software Components in 1968 at NATO Conference on Software Engineering [19]. Since then a lot of research interest has been aimed on this concept with practical implications to the software industry. It led to *component-based software engineering paradigm.*

The meaning of the term *component* depends on the context but there is a common consensus about its general features:

- Black-box part – user should not be concerned in implementation details
- Clearly defined boundary interaction points – interfaces and documented behaviour
- Reusable in different contexts (without a need of knowing or modifying its internals)

Collection of related abstractions, their semantics, and rules for composition of components creates together *component model*. Implementation of a *component model* is named *component system* [17]. Many component systems exist these days in both industry and academic sphere.

Component systems are aimed to provide runtime support for *components*. To make development of *components* (and software in general) more effective and less error-prone, a set of user tools can become helpful to automate various "bookkeeping" tasks and allow the developers last more resources on the creative part of development. It started with simple tools like *make* or *diff.* Nowadays there are many IDE tools with graphical user interface and whole range of supportive tasks including automatic code compiling, documentation searching, debugging, code high-lighting and error marking (with auto-suggestions for fixes of common errors like typos), version control systems access, etc. *Eclipse, NetBeans, Visual Studio,* and many others can be named as examples of such IDE tools. Applying the knowledge gathered by *User Interface Design* and Human-Computer Interaction areas; significant advancements have been accomplished in IDE tools as well as in user interfaces of applications for many other areas.

This thesis aims to provide at least basics of such IDE support for development of component based applications for the SOFA 2.0 component system[1].

1.1 Goals of the Thesis

The goal of this thesis is to design and to implement a prototype implementation of SOFA IDE – a tool for SOFA 2.0 distributed hierarchical component system.

[1] http://sofa.ow2.org/

SOFA IDE should enable component developers to perform more productive the development of applications composed from hierarchical components provided by SOFA 2.0 system by providing support for:

- Browsing the component elements contained in the component repository of SOFA 2.0 system
- Retrieving the component elements from component repository to Eclipse Workspace Project where they can be modified by editors supplied by Eclipse Platform and then committing them back to the repository
- Actions for creating new versions of component elements
- Providing a diagram editor for visual development of *hierarchical architecture* elements of SOFA 2.0 system

These goals as well as used terminology is further described in Chapter 2.

Prototype of SOFA IDE should be implemented as an Eclipse IDE plug-in.

1.2 Structure of the Thesis

Chapter 2 gives an analysis of the SOFA 2.0 system architecture, used technologies and appoints requirements for the prototype implementation of SOFA IDE plug-in. Analysis of Eclipse Platform and supportive technologies that can be used to develop SOFA IDE more effectively follows in the next sections.

Chapter 3 describes the solution of the prototype implementation and gives some alternative ways how it can be achieved.

Chapter 4 evaluates the achieved solution by practical demonstration of capabilities of SOFA IDE for developing a small SOFA 2.0 application in a *how-to* document style. Discussion about the achieved solution follows next.

Chapter 5 reviews work related to the SOFA IDE.

Chapter 6 concludes results of the thesis and describes ideas for future work which can not be achieved because of limitations resulting from the thesis format and range, and from no previous experience with SOFA 2.0 component system.

Appendix A describes how to download and install SOFA IDE plug-in.

Sections documenting the SOFA IDE itself are written with a practical reuse as SOFA IDE user and project documentation in mind. The style of these sections is tailored for this purpose as well.

2 Background

2.1 SOFA 2.0

The SOFA 2.0 system[2] is analysed in following sections from the components development perspective. An architectonic overview of SOFA 2.0 system is described in Section 2.1.1. Attention is then pointed mainly at SOFA 2.0 component model described in Section 2.1.2, because the purpose of SOFA IDE is to support development of components conforming to this model. Internals of SOFA 2.0 runtime environment are also particularly interesting but this topic is unfortunately out of scope of this work. A short overview of SOFA 2.0 runtime environment is provided in Section 2.1.3.

2.1.1 Architecture of SOFA 2.0 System

For purposes of this work we split SOFA 2.0 system into *runtime environment* part and *development* part.

Runtime environment provides required infrastructure for running distributed SOFA 2.0 applications. One instance of SOFA 2.0 runtime environment is also jointly denoted as a *SOFANode.* SOFA 2.0 is a distributed system and one SOFA Node can actually span multiple physical hosts.

[2] http://sofa.ow2.org/

Development part is used for development of new components or component based applications in SOFA 2.0 system.

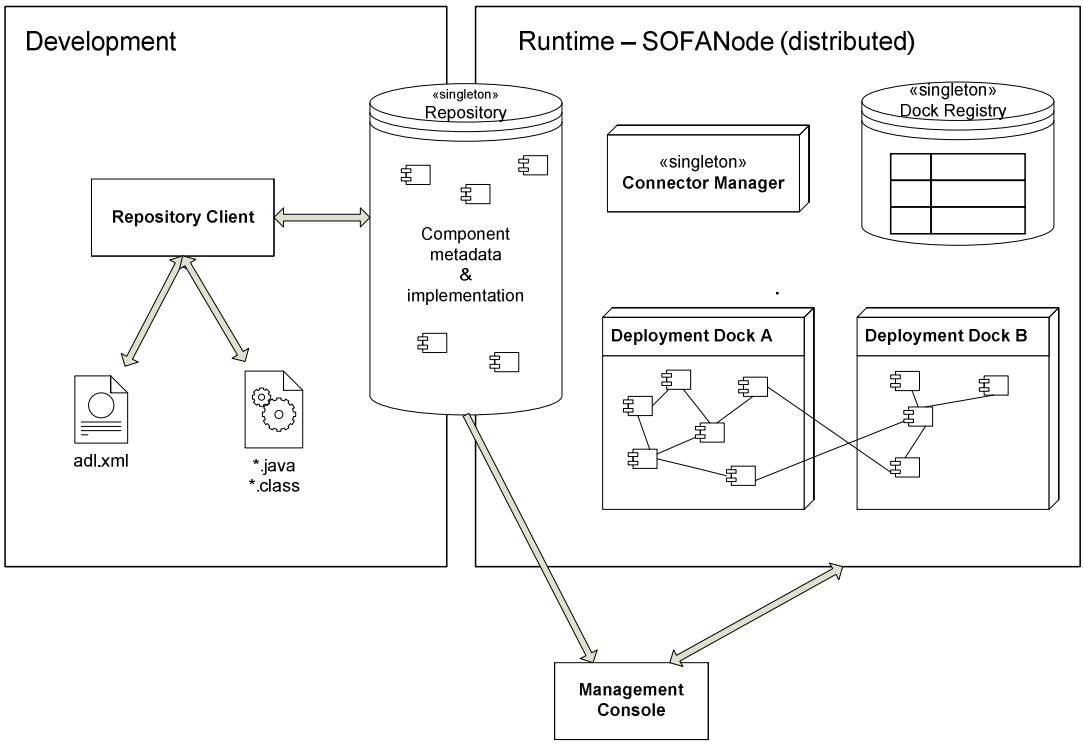


Figure 1 SOFA 2.0 Architecture Overview

2.1.2 Component Model of SOFA 2.0

One of unique features of SOFA 2.0 component system is a natural ability to (recursively) compose components from simple ones, so called *primitive components,* to complex hierarchies of *hierarchical components*.

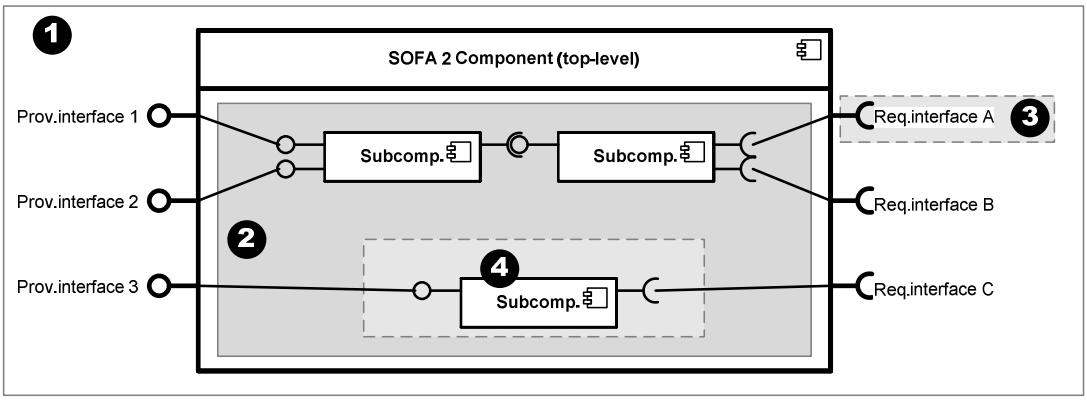


Figure 2 The anatomy of a hierarchical component (one nesting level displayed)

Elements of SOFA 2.0 components can be described by following basic building blocks:

Named entity, versioned entity (abstract classes) form base for primary component elements[3]. The Named entity contains a name attribute which must be unique within a given containment context. The name can be used to identify this entity within its contained context. The Versioned entity is derived from the Named entity. It contains a name and version number attributes. The doublet (name, version ID) must be unique within the given containment context. The Versioned entity can be referred to by specifying both, the name and the version ID (see below). If only the name is specified, then the latest *head* version of the entity in a repository with the given name will be resolved.

Version ID attribute The versioning model of SOFA components is based on the Distributed versioning model for MOF [15] suitable for distributed

[3] Component *entity* vs. component *element*: These terms are used as synonyms in context of SOFA 2.0 system. While report presenting the SOFA 2.0 metamodel [17] mentions *interface type element*, *frame element*, ...; the actual implementation of SOFA 2.0 model uses terms *NamedEntity*, *VersionedEntity* for names of base classes of other component elements. This thesis tries to adhere to the term *element*, but with object polymorphism in mind, terms *versioned* or *named entity* are also sometimes used.

repositories. Each version is identified by a pair consisting of the location ID (SOFA Node) and the version number. This schema preserves relations among versions of the Versioned entity (compare with opaque UUID [18] used e.g. by COM/DCOM). The repository provides functions for creating a new *Versioned entity* with an initial version ID as well as next and branched versions of the existing element with respect to the given originating version ID.

Frame (see Figure 22-1) is the main element of SOFA 2.0 model. It declares a black-box view of the component – how is the component seen from the outside by specifying a list of *provided* and *required interfaces*. The list of *provided interfaces* specifies services which are propagated by the component implementing the *frame* into the outer environment and other components can use them. The list of *required interfaces* specifies services which are required by the component implementing this *frame* and they must be provided by the environment. The *frame* is the *Versioned entity* contained in a repository on the top-level context.

Interface (see Figure 22-3) declares an end-point for creating connections among components at runtime. The interface is contained in the context of a frame. It specifies the name, name of *interface-**type***, communication style (RPC, SHM, streams…), connection type (normal / utility) and cardinality (single / collection) of *interface*. The communication style defines a type of connector used at runtime. Connectors in SOFA 2.0 are extensible and available communication styles depend on connectors presented in runtime. The *interface* is the *Named entity* which is contained in a context of a particular *frame.*

Architecture (see Figure 22-2) implements the component by specifying the name of Java class which implements all interfaces specified by the frame (primitive component); or by specifying sub-components (see Figure 22-4) with suitable frames and optionally specifies default architectures and default connections between sub-components

together and sub-components and interfaces of the top-level frame (hierarchical component). The architecture is the *Versioned entity* contained in a repository on the top-level context.

Interface Type defines the type of the interface by specifying a signature (the name of the type in the underlying language). The interface type is the *Versioned entity* stored in the top-level context of a repository (comp. *interface* stored within a context of a *frame*).

Assembly Descriptor specifies all architectures and connections among the whole hierarchy of sub-components which implement the given (top-level) architecture. The assembly descriptor can be generated fully automatically if specifications of default architectures and connections of sub-components which implement the given *architecture* are available in involved *architectures*. The assembly descriptor is the *Versioned entity* contained in the top-level context of a repository.

Element	*named / versioned*	*Description and properties*	*Used by*
Frame	yes / yes	Outer-side of component by listing interfaces of component: • provided • required	Architecture
Architecture	yes / yes	Inner-side of component by providing the implementation: • Hierarchical o Subcomponents (name, frame, default architecture [optional]) o Connections (optional) • Primitive (java class)	Assembly Descriptor
Interface-type	yes / yes	Type of interface by providing signature of Java interface	Interface
Interface	yes / no	Contained by Frame. Specifies particular interface of component by specifying: • Interface-type (reference) • Communication style (dependent on available connectors) o RPC o shared memory o messages (async) o streams o … • Connection type o normal o utility • Cardinality o single o collection	Frame, Architecture
Connection	no / no	Contained by Hierarchical Architecture (default connections) or Assembly descriptor (actual connections). Specifies interconnections between top-level component interfaces and sub-component interfaces as well as among sub-component interfaces.	Architecture (hierarchical), Assembly descriptor
Assembly descriptor	yes / yes	Specifies all connections and architectures of whole hierarchy of sub-components that implement given architecture.	(deployment of SOFA2 application)

Table 1 Elements of SOFA 2.0 components

2.1.2.1 Implementation Details about SOFA 2.0 Repository

Component elements (meta-data as well as code) are stored in a *repository* during both development and run time phases as the Versioned entities. Core of the repository is defined and generated using the EMF framework (see Section 2.4) extended to support remote access to resources[4] and façade layer for the versioning support, because versioning is not directly integrated into the EMF framework by itself. To allow seamless development of new components with temporary inconsistencies during the development phase, a repository cloning technique is available. Before a developer starts developing of new components, he clones the existing repository, sets a separate development environment, and then he can freely perform changes and tests on this clone. The original repository stays unaffected by this process. The changed clone is merged back at the end of the development and merging process, ensuring that only consistent changes can be merged [7].

2.1.3 SOFA 2.0 Runtime Environment

The implementation of the runtime environment of SOFA 2.0 is based on an experience with implementation of the previous version of the SOFA system (SOFA 1) and is designed to address advanced features of the SOFA 2.0 component model like nesting of components, connectors for different communication styles, etc. The SOFA 2.0 runtime environment also allows for dynamic reconfiguration of *hierarchical* components at runtime in a managed way.

The main parts of the SOFA 2.0 runtime environment are described in Table 2. More detailed analysis of the SOFA 2.0 runtime environment is out of scope of this work. SOFA IDE interacts only with the *repository* part of an environment. Some description of the SOFA 2.0 runtime environment and its

[4] Using *WinStone Servlet Container* (http://winstone.sourceforge.net/) and *Commons HttpClient* (http://hc.apache.org/httpclient-3.x/index.html).

support for advanced features of a SOFA 2.0 model can be found in ,[6, 7, 8]. A dynamic reconfiguration of hierarchical components is also described in [16].

Runtime part	*Description*
Repository	Stores meta-data about components (the versioned entities above) as well as their implementations (as jar archives). Provides a necessary infrastructure for remote retrieving and storing components in the registry. Single instance per SOFA Node (see Section 2.1.2.1).
Connector	Provides implementation of connections among running components.
Deployment dock	A container for launching components with necessary infrastructure. A deployment plan is used to specify which components should be launched on which dock. Involved docks retrieve the meta-data and the implementation of components directly from the repository.
Dock registry	Registers all deployment dock instances existing within SOFA Node. Single instance per SOFA Node.
Connector manager	Responsible for connecting units of connectors together. Single instance per SOFA Node.
Management Console	User interface for management of running SOFA 2.0 component based applications as well as the run-time environment of SOFA 2.0 itself. See Section 5.1.1 which mentions a *SOFA 2.0 Management Console* application for further details.

Table 2 Quick overview of SOFA 2.0 Runtime

2.2 Current Way of Development of Component Based Applications in SOFA 2.0

Developers access development *repository* with *Cushion* – a Java-based command line client to retrieve (checkout) meta-data and Java code of component elements from repository to local *working directory* as well as to put (commit) changes back to repository. To make editing easier, Cushion translates the meta-data from EMF model to "human-friendly" form serialized into set of XML files with simple structure stored in the *working directory.* The *working directory* has a defined layout which uses subdirectories with names equal to underlying element names. Inside of these subdirectories is created the actual XML file with translated meta-data about component element. For

interface type and *primitive architecture* elements is additionally created *code* subdirectory holding the Java sources related to given component element. The version ID is not contained in the subdirectory name (which reflects component element name), nor in the content of XML file with meta-data about component element. Cushion stores this information together with names of component elements retrieved from repository in the root of working directory (file `_cushion.cfg`). Because of this, multiple versions of one component element cannot be put into working directory. This is not a big constrain in practice, because developers are working with just one version of a component element at most time. In special cases, a separate working directory can be easily created.

2.2.1 Call for Improvements

The general concept of *working copy* proposed by Cushion approach is found to be useful for development of component based applications for SOFA 2.0. However, to increase productivity and make the development less error-prone, there is a demand after support of this concept in an IDE tool with graphical user interface for exploring content of the repository with actions for retrieval and posting of component elements, editing the content of working directory as well as an ability to represent and edit the *hierarchical architectures* visually in diagrams similar to UML 2.0 component diagrams (see Figure 22) to naturally reveal used sub-component instances as well as default interconnection of their interfaces.

The goal of the thesis is to research possibilities and develop a prototype implementation of this tool, and to create an environment for further work in this area.

2.3 Eclipse Platform

The Eclipse[5] is an extensible development platform for tool integration with mature technologies composing the core Eclipse platform and a wide range of tools built around it with ability for an easy extensibility. The Eclipse is also a set of community-based projects maintaining the core technology and tools hosted by Eclipse Foundation with a diverse range of contributors. The Eclipse Java IDE is probably the most famous result of this effort. Not only a miracle but also well applied principles of a component-based development and many reusable components available within the Eclipse make the Eclipse Platform and extensions to it so successful. [9, 11, 12, 13]

2.3.1 Architecture of Eclipse Platform

The architecture of Eclipse Platform (Figure 3) is based on plug-in contributions – structured bundles of code and data. Plug-in contributes function to the system. A contribution of the plug-in can be performed in the form of a code library (public Java classes), platform extensions to extension-points, or documentation. Plug-ins can define extension-points by themselves. An Extension-point is a well defined place for adding of functionality by another plug-ins. The platform itself is composed by a micro-kernel for loading plug-ins and a set of standard plug-ins providing a core platform functionality. Platform plug-ins are naturally organized to platform sub-systems. The platform SDK adds plug-ins providing a useful functionality for developing of new plug-ins [11].

[5] http://www.eclipse.org/

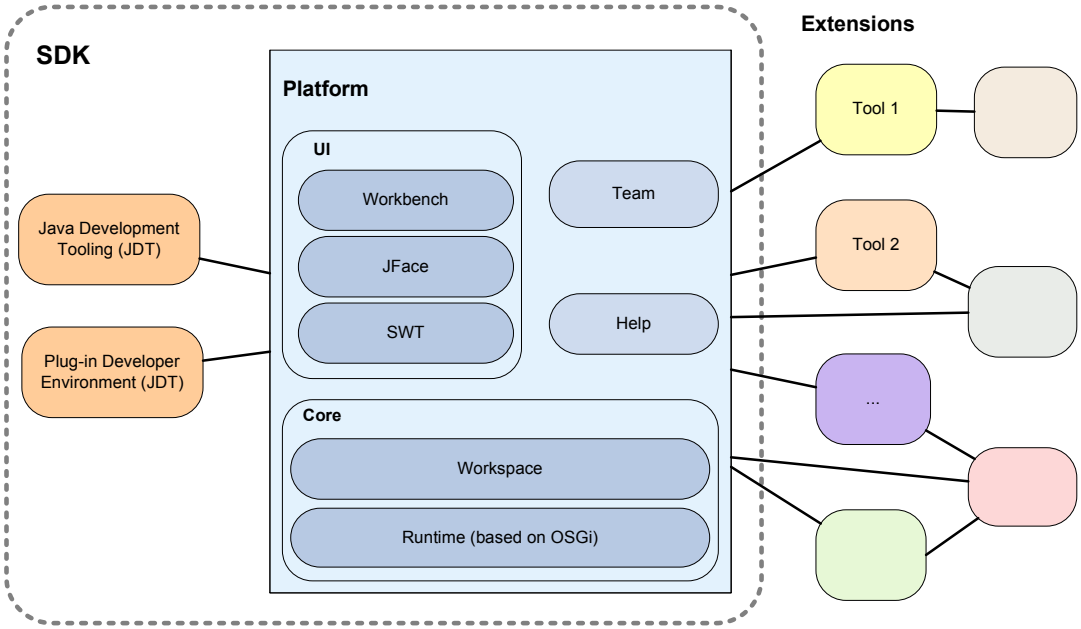


Figure 3 The Architecture Overview of the Eclipse Platform

Core – Runtime is a core part of the whole Eclipse Platform. It provides a runtime engine for bootstrapping and dynamic loading of plug-ins. The plug-in loading in *runtime* is implemented by using the *OSGi services model*[6]. Every Eclipse plug-in is effectively *OSGi bundle* (with dependencies on the Eclipse Platform). Plug-ins are loaded lazily, so there is no memory penalty for installed plug-ins which are not used. Each Plug-in is described by itself by providing a *plug-in manifest* containing meta-data about the given plug-in. *Runtime* maintains the registry of installed plug-ins, but the plug-in will not be activated until its function is not needed according to the current activity of a user.

Core – Workspace (***Resources*** plug-in) together provide a resource management for platform workbench organized projects, folders and files, commonly referred to as resources[7]. The *Resources* plug-in

[6] See Eclipse Equinox Project (http://www.eclipse.org/equinox/).

[7] Eclipse resource is a proxy [14] to a real file-system resource. It provides loose-coupling to the underlying file system resources (e.g. they even do not have to exist). It is also not required that tree organization of the Eclipse resources has to strictly reflect the organization of the

provides a functionality required for a management of these resources. The resources are organized around a resources tree model. The *Workspace* is a root of this model. *Projects* reside at the next level and both, the *folder* and the *file resources* are organized within *projects*.

UI – Standard Widget Toolkit (SWT) is a low-level part of Eclipse GUI. SWT provides platform-independent Java API for creating GUI widgets. The Java Part of SWT is light-weight for performance and tight OS GUI integration reasons and most of SWT internals are implemented natively for each supported OS platform.

UI – JFace makes many GUI programming tasks easier by extending SWT with JFace viewers / content providers, Actions / Contributions, GUI resource registries, various standard dialogs and field assist service API abstractions.

UI – Workbench is a "GUI container" plug-in providing multiple extension points allowing other plug-ins to contribute to menu and toolbar actions, drag & drop, views and editors to be revealed on the *Eclipse Workbench*. The main window which user sees when he starts the "Eclipse" application is maintained by the *workbench* plug-in.

Help plug-in integrates help functionality to the Eclipse. It provides extension points which can be used by other plug-ins to contribute to documentation in a form of browsable books.

Team Support provides a generic functionality that can be used by other plug-ins which implement support for a team programming, e.g. accessing version control systems. (Subversion, CVS, …)

---

underlying file-system resources. A common practice is that the underlying directory containing meta-data about a workspace root is created separately from directories with projects.

SDK – Java Development Toolkit provides tools for development of Java applications like source-code editing, compiling, debugging, refactoring, etc.

SDK – Plug-In Developer Environment further extends the functionality of the JDT plug-in by supplying specialized tools required for development of the Eclipse plug-ins.

The actual structure of *platform* plug-ins is more fine-grained. Previous paragraphs describe only main elements representing parts of the Eclipse platform functionality. See the Eclipse documentation [11] for a complete reference.

A lot of extensions providing an additional functionality to the Eclipse are built on the top of *platform* plug-ins. We can mention e.g. *Web Tools Project*[8] extending the Eclipse platform with tools for developing of Web and Java EE applications, or *PyDev*[9] which enables IDE for developing of Python and Jython[10] applications. Also the EMF and GMF projects mentioned in Sections 2.4 and 2.5 provide a functionality, frequently used by other Eclipse plug-ins.

2.4 Eclipse Modeling Framework

The Eclipse Modelling Framework (EMF) is part of *Eclipse Modeling project*. The Eclipse Modeling project concentrates model-based development technologies within Eclipse community and provides variety of frameworks, tools and standard model implementations to support of model-based development[11].

[8] http://www.eclipse.org/webtools/

[9] http://pydev.sourceforge.net/

[10] Jython is an implementation of the Python language in Java (http://www.jython.org)

[11] Eclipse Modeling project home-page: http://www.eclipse.org/modeling/

EMF is a Java framework as well as a code generation tool for an automated creation of domain models using a formal model definition as an input. If used for development of domain-specific language [26], EMF can be used for implementation of abstract syntax.

It consists from two fundamental parts: *EMF Core* and *EMF.Edit* framework.

EMF Core framework provides a code generation and runtime support with Java classes which implement the model.

EMF.Edit framework is based on and extends *EMF Core* framework by providing code generation and runtime support for adapter classes for viewing and command [14] based editing of models with undo / redo support, and simple tree-based editors which use JFace API provided by Eclipse Platform for implementation of user interface.

The EMF model generator uses XMI [23] as a canonical form of a model definition. The XMI form of the model can be obtained by multiple ways including:

- Export from UML class diagrams created by modeling tools supporting export to XMI
- Annotations to Java Interfaces with model properties
- Emfatic[12]
- XML schema by describing the serialized form of the model

The EMF generator uses this input for creating of corresponding Java sources with interfaces and classes with implementation of this model. The interesting part is that a user can edit generated sources and the changed portions will be preserved by the EMF generator during a regeneration of the model, even if the

[12] See Emfatic Language for EMF Development (http://www.alphaworks.ibm.com/tech/emfatic) for details about Emfatic project.

model is changed (user has to modify a customizations affected by changes in the model to conform the new model after this regeneration). The generated model uses the EMF meta-model implementation (Ecore) to provide meta-data about structural and behavioural features of the model, generic reflective API to the model, and change notification support. EMF also provides a persistence of models. Default implementation uses XMI or XML schema–based serialization. Multiple projects extend EMF by adding relational databases persistence, query, or validation support. See home-page of EMF project for further details.

2.4.1 Implementation of EMF Metamodel (Ecore)

The Ecore itself is based on the MOF specification [22]. However, the Ecore is not focused on the meta-data repository management like MOF, rather on a tool integration. So the Ecore implements just portions of the MOF specification which is practical for a usage within EMF. The simiarities between EMF and MOF can be found in a way how are the structural and behavioural features of the model specified, or in a generic reflective interface to access the model. This approach leads to an optimized and widely applicable implementation. [2, 4, 20]

2.4.2 EMF and SOFA

SOFA 2.0 repository model is generated with EMF framework from the meta-model definition described in UML class diagrams. SOFA 2.0 implements customized serialization of EMF model resources allowing remote access to the model with transparent lazy-loading and caching of requested model parts and adds façade layer for management of multiple versions of elements (versioned entities) stored in the repository. Because EMF does not support versioning natively, this feature comes with down-side that standard *model factory* provided by Ecore cannot be used for creating versioned entities directly and the façade layer has to be used. This hinders simple usage of standard tools integrating with standard EMF models for editing the repository model in

cases when a new versioned entity should be created and custom approach is needed. [2, 4, 20]

2.5 Graphical Modeling Framework

Graphical Modeling framework[13] (GMF) is also a part of Eclipse Modeling project. It provides code generation facility and runtime support for development of graphical editors. GMF aims to create a bridge between Graphical Editing Framework (GEF) [20] and EMF technologies. In the context of DSL, GMF can be used for implementation of graphical concrete syntax.

GEF allows creating diagram editors. It uses Draw2D framework (based on top of SWT) to perform actual drawing and adds logic for editing support. However, it is model-agnostic and virtually any model suitable for representation in graphical form can be used with GEF. It is responsibility of the user to implement the connection between GEF and the used model. On the other hand, EMF allows creation of structured models based on Ecore. It started to be common to use both EMF and GEF together to implement diagram editors. Common set of similar problems involving connection of EMF model to GEF diagram API, which should be repetitively solved for every diagram editor built on top of EMF and GEF, has been revealed. The bridge provided by GMF links EMF and GEF together for easier creation of graphical editors that display data stored in EMF models.

GMF generates this functionality in similar way, how EMF generates implementation of model according to its definition. This process is outlined in Figure 34. GMF generator requires user to define a set of models defining the visual aspects (graphical definition), tools for manipulating (tooling definition) and notation mappings (mapping model) of EMF-based domain model, and generates editor implementation for this model into a Diagram Plug-In project. Additionally it uses *notation* model for storing visual data (position of shapes,

[13] http://www.eclipse.org/modeling/gmf/

connection routes, etc.) about content of underlying domain model. GMF generated code can be hand-customized in a way similar to EMF – the changes will be preserved after regeneration of the code. GMF-generated code does not use GEF directly. It uses *GMF runtime* framework that enables creation of separate extension plug-ins with further functionality customizations of generated diagram plug-in by extending *extension-points* created by *GMF runtime.* [1]

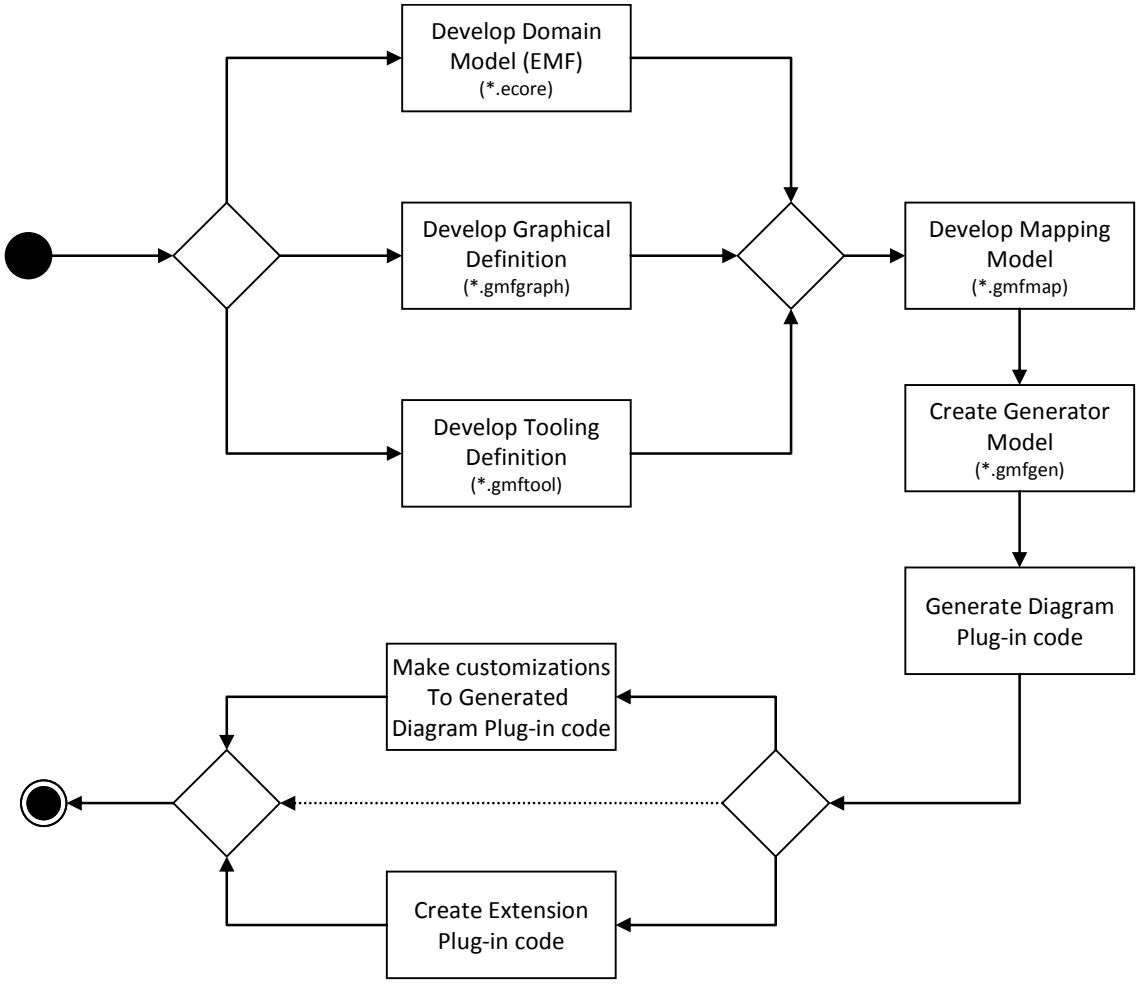


Figure 3 Process of developing a diagram editor with GMF framework

3 SOFA IDE

We partition functionality of proposed SOFA IDE plug-in into four parts / subsystems: *Repository browser, SOFA Projects manager, Actions for interacting with repository,* and *Editor for architecture diagrams*, so we can talk about each part separately. These parts integrate well together, though.

3.1 Repository Browser

Repository Browser part displays content of SOFA 2.0 Repositories in tree viewer. It allows adding of multiple repositories as well as the displayed elements can be retrieved from repository to SOFA Project (see Section 3.2) by Checkout action (see Section 3.3).

Repository content is provided in form of EMF model (see Section 2.4). SOFA IDE uses functionality provided by EMF.Edit adapters to adapt the repository model to a form suitable as an input to JFace tree viewer implementation.

SOFA 2.0 Repository is remotely accessed with HTTP protocol to provide the component elements. The retrieval method is encapsulated into Repository Model implementation and user of the API just provides the address of Repository to initialize a *RepositoryAgent* object and then the Repository Model can be transparently accessed with the *RepositoryAgent* object. Content of component elements is lazy-loaded by on-demand resolving of proxy objects and transparently retrieved from remote repository. This approach conflicts with the usage of EMF.Edit adapters, because they examine

the EMF model as the underlying content provider asks for portions of it and remote resources are accessed in UI thread that is blocked and application "freezes" in these moments. SOFA IDE deals with this problem by providing a level of indirection into model access code. When the top-level node of a repository is expanded by user at the first time, a *pending repository* object with "Loading…" label is provided to the user. The content of the repository is retrieved in background job by simply traversing the portions of model that will be accessed by EMF.Edit adapters when displayed in Tree Viewer. The model proxies are resolved and initialized with actual model content. Then the auxiliary object is replaced with the actual preloaded content of repository and a change notification is fired to perform refresh of involved viewers.

3.2 SOFA Projects Manager

The concept of *working directory* described in Section 2.2 introduced by Cushion almost fits to the *project* concept used by *Eclipse workspace* (see Section 2.3.1). However, *Resources* plug-in in *Eclipse Platform* provides just a generic lookout on resources: 'This is a directory; this is a file; that is a project…' The purpose of *SOFA projects manager* is to reveal and manage 'sofa-centric' information about resources contained in *SOFA Project:* 'This is frame in version xyz; that architecture is from foreign repository and is locked; that project is associated with SOFA 2.0 repository running at localhost…'

To achieve this purpose, SOFA Projects manager builds a custom model with SOFA related meta-data about the resources contained in the projects with enabled *SOFA Project Nature*[14]. Information from this model is then used

[14] *Project Nature* is a generic approach supported by Eclipse how to distinguish between different types of projects contained in workspace. Java projects have assigned *Java Nature;* Plug-in development projects have assigned *PDE Nature* as well as *Java Nature* because plug-ins for Eclipse are primary built in Java, etc. Plug-ins can register activation of a Java class when a project with specific *nature* is opened or created. In the case of SOFA IDE, the SOFA Projects Manager registers it implementation class with SOFA Project Nature and is started by

by multiple other parts of the SOFA IDE. E.g. there is a specialized label decorator which reveals type and version information about element contained in SOFA Project by decorating label and icon of its directory shown in *Project Explorer* view; or by *actions for repository interaction* (see Section 3.3) to determine the URL of repository associated with selected SOFA Project.

The directory and file structure of SOFA Project stays backward compatible with the layout used by Cushion (with exception of *architectures*, where a format optimized to use with the *Architecture Diagram Editor* is used). SOFA projects manager is based on Cushion's *CushionConfig* implementation but the SOFA IDE extends this concept with adding support for convenient retrieval of element type information, change notification, and adapter support to adapt[15] generic Resource objects to SOFA Project model items. SOFA Projects Manager is a core part of SOFA IDE plug-in and provides functionality to other parts of SOFA IDE. It could be usable even by other plug-ins to further extend functionality of SOFA IDE.

3.3 Repository Interaction Actions

Additionally to Repository Browser, SOFA IDE provides actions which implement retrieval of component elements from repository to SOFA Project, posting the changed elements back to repository, as well as creation of new versions (initial, next, branch) of versioned component elements. They can be executed from appropriate context menus (checkout from Repository Browser, commit from context menus of SOFA Project resources, new from *New Wizards* menu…). These actions serve the same purpose as Cushion, but the underlying implementation of Cushion cannot be used directly and has to be refactored to fit needs for usage within SOFA IDE.

Eclipse automatically (only) when a project with SOFA Nature is opened or created in workspace conforming to general lazy-loading of plug-ins.

[15] Adapter pattern (see [14]).

3.4 Editor for Architecture Diagrams

As mentioned in Section 2.1.2, SOFA 2.0 system provides support for component nesting to form hierarchical components. This can be effectively visualized with diagram similar to UML 2.0 component diagrams (see Figure 22).

Building of such diagram editor from scratch is a challenging task. To make this process more effective, appropriate supportive technologies and tools can be applied to speed-up this process. We use the GMF framework (see Section 2.5) as such technology. Having no previous experience with GMF, this task is still not easy. For practical reasons, we have created a custom-tailored model for *architecture* element optimized for GMF without a need to provide many customizations in GMF-based diagram drawing code. In this case it is easier to provide additional transformation within known context in checkout / commit actions, and it allows us to get the proof-of-concept solution sooner.

To preserve coordinates of nodes and links, this diagram model is attached to the model of *architecture* stored in *repository* at commit as an annotation. This annotation is used as a hint at check-out from repository to recreate the architecture diagram model. If the *architecture* has been changed and the diagram model presented in annotation is out of date, changes are automatically applied using the *architecture* meta-data from *repository* as a master. Missing coordinates are computed by auto-layout algorithm provided by GMF. This is also the case when an existing *architecture* is opened in the diagram editor for the first time – there is no annotation with diagram model.

The resulting drawing provided by default GMF auto-layout is not very usable and user has to untangle it manually at the first time. The auto-layout of an architecture diagram is out of scope of this work.

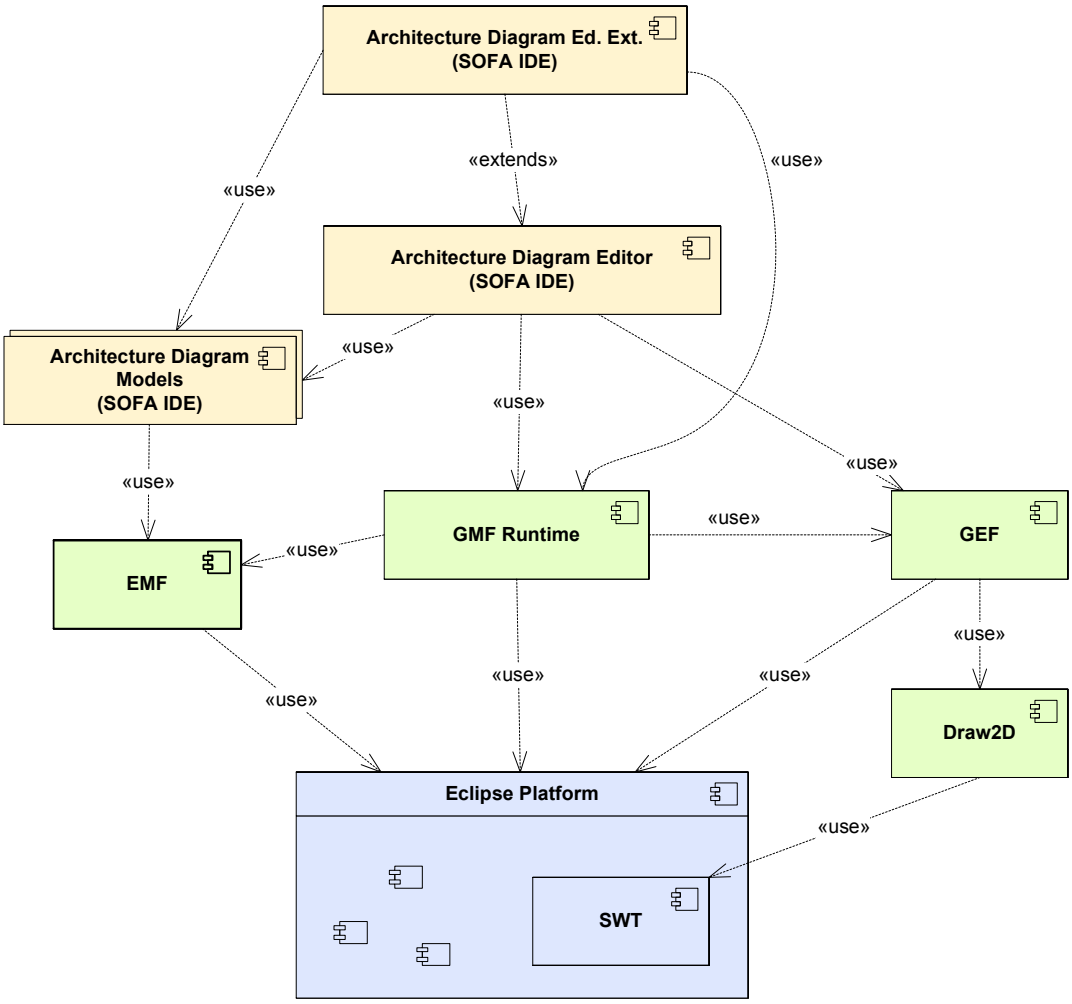


Figure 4 Dependency Structure of Architecture Diagram Editor

4 Usage of SOFA IDE

The following section demonstrates the capabilities of a newly created SOFA IDE Eclipse plug-in by creating a simple SOFA 2.0 application[16]. Section 4.2 gives a discussion about (dis-)advantages of the SOFA IDE Plug-in compared to a Cushion approach.

4.1 How to Create SOFA 2.0 Application with SOFA IDE

We are going to create a simple SOFA 2.0 application to demonstrate capabilities of the SOFA IDE plug-in for development of SOFA 2.0 components.

[16] The LogDemo application is the same as presented in the Cushion tutorial *How to create SOFA 2 application in several steps* (http://sofa.objectweb.org/docs/howto.html).

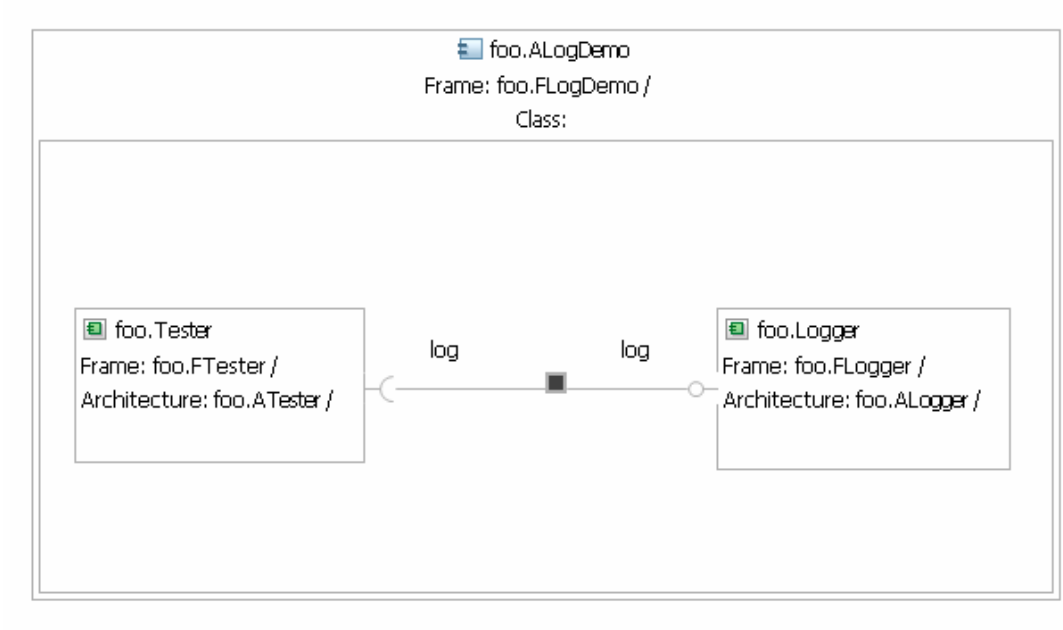


Figure 5 Architecture of the LogDemo Application

The application will consist from three components: *LogDemo* (hierarchical), *Tester*, and *Logger* (primitive components). The architecture of this application is shown on Figure 5. The *Tester* component requires an interface *ILog* which is provided by the *Logger* component and is connected to the *Tester* component. Finally, both components are encapsulated into the *LogDemo* composite component.

4.1.1 Accessing SOFA Repository

We are going to reveal the SOFA Repositories view on a *Workspace*. Open the Show View dialog by selecting the menu **Window > Show View > Other...** and select the SOFA Repositories view in the section **SOFA**. This view will be revealed on the Eclipse **Workspace** after a confirmation and it allows for browsing of a content of connected *SOFA repositories* (see Figure 6).

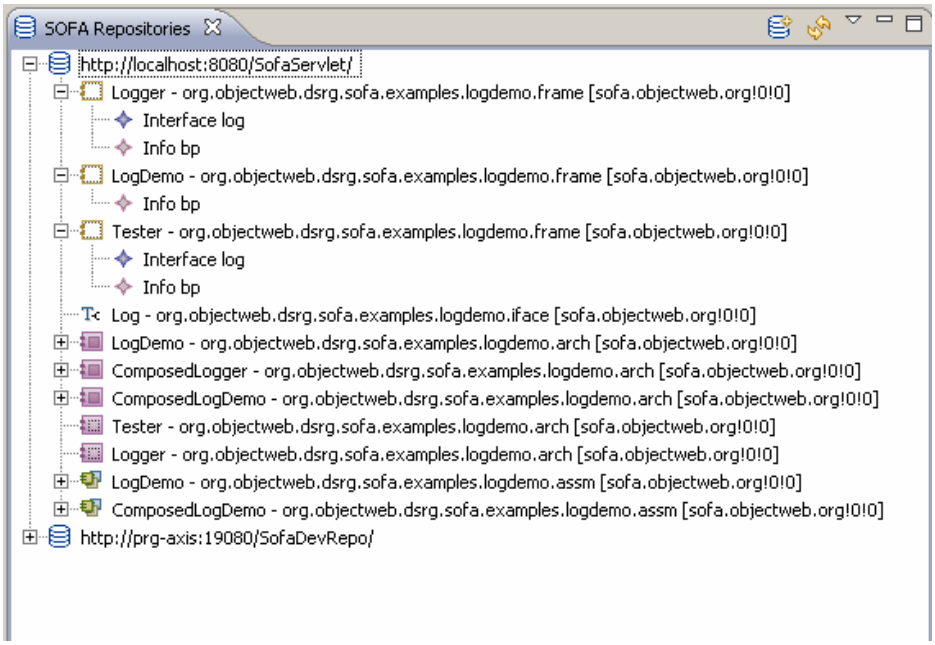


Figure 6 The SOFA Repositories View

Now, we will add a new *SOFA Repository Location* into the SOFA Repositories browser. Select the Add SOFA Repository tool on a *local toolbar* of the SOFA Repositories view. The **Add SOFA Repository** dialog will be opened. Type `http://localhost:8080/SofaServlet` into the **Repository URL** field and confirm with the **Finish** button.

> ***Tip***
>
> *The **Check Connection** button can be used to quickly reassure that the SOFA repository is running at the specified URL. The **Browse** button can be used to select an existing repository location for adding multiple repositories with similar the URL.*

The content of the added SOFA Repository will appear in the SOFA Repositories view. Multiple repositories can be added into this view as well (see Figure 7).

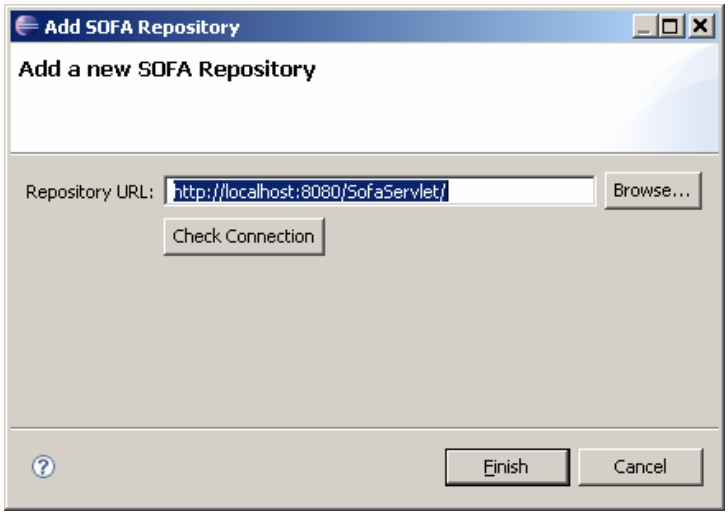


Figure 7

4.1.2 Creating a New SOFA Project

Firstly, one has to create a new Java Project by selecting File > New > Java Project in the main menu. Type `LogDemo` into the **Project Name** field and confirm it with the **Finish** button. The *LogDemo* project should appear in the Project Explorer view.

Then we will convert the *LogDemo* project to the *SOFA Project* type and we will assign a SOFA repository to it. Open the context menu of the *LogDemo* project by a right-click on its label. Select SOFA > Add SOFA Nature in the context menu. The Add SOFA Nature Project dialog will be opened similar as the Add SOFA Repository dialog mentioned in Section 4.1.1. The difference is that *now we have to assign the SOFA repository which will be used with the given LogDemo project* not only in the SOFA Repositories browser. You can use the **Browse** button to select the **Repository URL** address http://localhost:8080/SofaServlet/ and to specify the repository assigned with the *LogDemo* project. Then simply confirm it with the **Finish** button.

Tip

SOFA projects can be easily distinguished from other projects by their icon containing a red "S" as an overlay in the upper right corner. 

4.1.3 Creating Component Elements

The LogDemo application consists of following component elements:

Name	***Type***
foo.ILog	interface type
foo.FLogger	frame
foo.FTester	frame
foo.FLogDemo	frame
foo.ALogger	primitive architecture
foo.ATester	primitive architecture
foo.ALogDemo	hierarchical architecture

New component elements can be created with the SOFA IDE by using *New Wizards* – a standard way how new Resources are created in the Eclipse. Select the menu File > New > Other… to reveal a dialog with tree structure of resource types that can be created in the Eclipse. The SOFA elements are listed within the node **SOFA: Architecture**, **Frame**, and **Interface Type**. Select appropriate item and press **Next** button. A dialog for creating a new component element of selected type will be shown (see Figure 8).

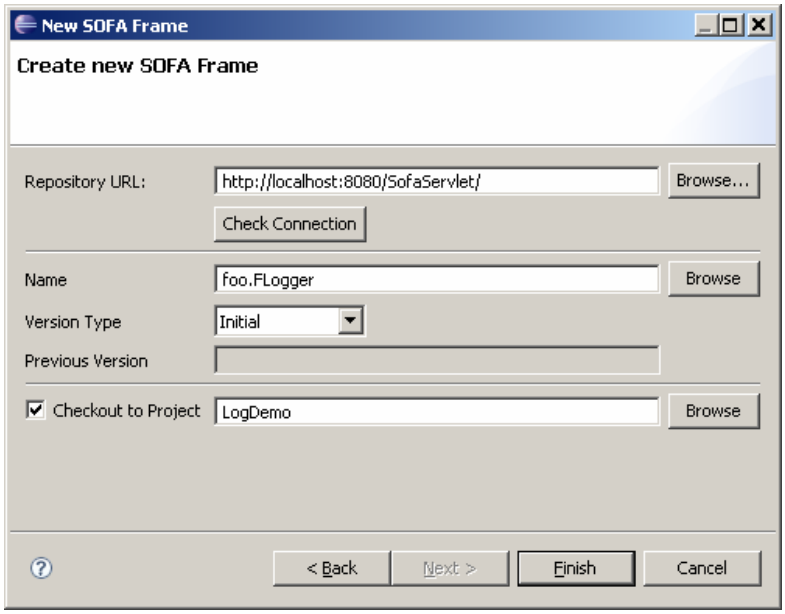


Figure 8 Creation of a new frame element

The **Repository URL** field specifies the location of the component repository where the new element should be created.

The **Name** field specifies the name of the new component element. The **Version Type** combo box (Initial, Next, Branch) specifies the kind of creation operation. When the *initial* is selected, a new component element in the initial version will be created in the repository. The name of this component element has to be unique and no existing component element can hold the same name. When the *next (branch)* is selected, a new version (branch) of existing component element is created. The field **Previous version** should specify the originating version from which the new version should be created. **Browse** button can be used for lookup of existing component elements for this purpose.

If the **Checkout to project** field is checked, the appropriate SOFA Project should be selected and the new repository element is automatically checked to the SOFA Project additionally to creating it in the component repository. The SOFA Project has a fixed directory structure known from the Cushion workdir: *<element name>/adl.xml* (see Section 2.2). For the interface type and (primitive) architecture elements, additional *code* subdirectories are created within the *<element name>* directories to hold the implementation classes of these elements.

4.1.3.1 Interface Types and Frames

Create a new *interface type* named *foo.ILog* and also checkout it to the *LogDemo* project. Open the *adl.xml* file of the *foo.ILog* interface type (placed in *foo.ILog* subdirectory) and add a `signature="foo.ILog"` attribute to the `itf-type` XML element:

```
<?xml version="1.0"?>
<itf-type name="foo.ILog" signature="foo.ILog" />
```

Then create the *Java interface* `foo.ILog` in the foo.ILog/code/foo/ILog.java:

```
package foo;
public interface ILog {
  void log(String message);
}
```

Finally commit the changes back to the repository by selecting **SOFA > Commit** action from the context menu of `foo.ILog` folder in the **Projects Explorer** view**.** The changed meta-data and the Java code will be transferred to the repository.

Create frames *foo.FLogger, foo.FTester*, and *foo.FLogDemo* in similar way. However, frames do not have any Java code that implements them, only the *adl.xml* file is located in their directories. The content of the adl.xml files of these frames is shown below:

`foo.FLogger/adl.xml`:

```
<?xml version="1.0"?>
<frame name="foo.FLogger">
```

```
  <provides name="log" itf-type="sofatype://foo.ILog"/>
</frame>
```

foo.FTester/adl.xml:

```
<?xml version="1.0"?>
<frame name="foo.FTester">
  <requires name="log" itf-type="sofatype://foo.ILog"/>
</frame>
```

foo.FLogDemo/adl.xml:

```
<?xml version="1.0"?>
<frame name="foo.FLogDemo">
</frame>
```

The *foo.FLogger* frame provides an interface *log* of type *foo.ILog*. The *foo.FTester* frame require an interface *log* of type *foo.ILog* to be connected. The *foo.FLogDemo* frame is a top-level frame of the LogDemo application and does not provide or require any interfaces. Commit the changes back to the repository and the *architecture* elements conforming to given *frames* can be created.

4.1.3.2 Creating Architectures with Architecture Diagram Editor

New *architectures* can be created in a similar way how the *frames* and *interface types* are created by using the *New Wizards.* The *SOFA* → *Architecture* should be selected. The LogDemo application consists of three architectures: *foo.ALogger, foo.ATester,* and *foo.ALogDemo.*

The *foo.ALogger,* and *foo.ATester* architectures are primitive. Just fill the name of the frame declaring the architecture (*foo.Flogger,* and *foo.FTester*), and the fully qualified name of the class implementing them into the header of the diagram and do not create any sub-components within them. The Java code

implementing these *architectures* is listed below and should be placed to appropriate *code* subfolders.

ALogger.java:

```
package foo;
public class ALogger implements ILog {
  public void log( String message ) {
    System.out.println("LOG: " + message);
  }
}
```

ATester.java:

```
package foo;
import org.objectweb.dsrg.sofa.SOFAClient;
import org.objectweb.dsrg.sofa.SOFALifecycle;
import org.objectweb.dsrg.sofa.SOFAThreadHelper;

public class ATester implements SOFALifecycle, Runnable, SOFAClient
{

  boolean end = false;
  ILog logger = null;

  // Implements method from the SOFALifecycle interface.
  // Called during instantiation of the component.
  public void start() {
    Thread t = new Thread(this);
    t.start();
  }

  // Implements method from the SOFALifecycle interface.
  // Called as a notification that component will be stopped
  public void stopping() {
  }

  // Implements method from the SOFALifecycle interface.
  // Called during stopping of the component
  public void stop() {
    end = true;
  }

  // Thread of the component (java.lang.Runnable interface)
  public void run() {
    while (!end) {
      logger.log("Hello world!");
      try {
        Thread.sleep(5000);
      } catch (Exception e) {}
    }
  }

  // Implements method from the SOFAClient interface
  // Called during initialization of the component
```

```
  public void setRequired(String name, Object iface) {
    if (name.equals("log")) {
      if (iface instanceof ILog) {
        logger = (ILog) iface;
      }
    }
  }
}
```

The *foo.ALogDemo* architecture is composite. Fill the name of the frame implementing the architecture (*foo.FLogDemo)* and draw two sub-components in the diagram of the *foo.ALogDemo* architecture with names *foo.Logger* and *foo.Tester*. See Figure 5. Fill the *frame* names of these sub-components (*foo.Flogger* and *foo.FTester*). Specifying the architecture names for sub-components is optional. They can be specified at deployment phase in the *assembly descriptor.* Select nodes of these sub-components and reveal a context menu by right-click. Activate the **Synchronize Interfaces** action. This action will query frames of the sub-components stored in the repository (make sure that they are commited) and reveal all required and provided interfaces on sub-component diagrams[17].

Create a *connector* point and connect the *log* interfaces together. Save and commit changes back to the repository in a similar way as committing frames. The *code-bundles* containing Java code of the primitive architectures are automatically created and committed when the primitive architectures are committed, similar to the interface types.

4.1.4 Next Steps

We have successfully created components for our LogDemo application. To execute the application, an assembly descriptor and deployment plan has to be created with a configuration of deployment docks and the application has to be deployed to a runtime environment using this deployment plan. These deployment tasks are not covered as a part of this thesis. They can be

[17] This is not automatic in prototype implementation, because the repository is running remotely.

performed by the *Cushion* or newly developed *M2Console* plug-in for the Eclipse [10].

4.2 SOFA IDE vs. Cushion

Compared to an old Cushion command-line client, the SOFA IDE plug-in brings a lot of comfort and productivity improvement for component developers using the SOFA 2.0 system. And there is a big advantage from synergy of integration with the Eclipse Platform itself. The Eclipse with many plug-ins available is recognized as a pervasive IDE for a software development of these days, distributed systems included.

The Cushion with its command line interface can be useful for special scenarios like “only SSH access is available”, or for automation of various tasks including testing, deployment and installation scripts, or disaster-recovery actions as an additional tool to the *repository cloning* technique (see Section 2.1). These cases are well described e.g. in [21]. This functionality is not available with SOFA IDE as a visual tool. The *IBM DB2 Control Center* – a tool for administration of IBM DB2 databases – can be mentioned as an example of this approach. It allows its users to get and review SQL commands for any task they perform.

5 Related Work

5.1.1 SOFA 2.0 Management Console

The SOFA 2.0 Management Console[18] *(SOFA2MConsole)* is a complement tool to *SOFA IDE*. While *SOFA IDE* is dealing with the development part of components for SOFA 2.0 system, the *SOFA2MConsole* is used for managing SOFA 2.0 runtime environment. It allows to start and to stop applications, monitoring their state, monitoring the state of deployment docks and visualizing communication of running components among deployment docks according to a specific deployment plan.

The SOFA2MConsole also uses the Eclipse Platform for an implementation but it is not developed as a Plug-In, rather it is a stand-alone application based on the Eclipse Rich Client Platform Project (RCP). The main difference between Eclipse Plug-Ins and RCP based Applications is in the way how they are used. The Eclipse Plug-Ins are running in context of the Eclipse Workbench and they use View and Editor parts to reveal their functionality. Their intended usage is in case when the *IDE* metaphor with workspace, project folders and potentially many plug-ins running side by side is desirable. The RCP application "owns" its main window and the whole layout and a function is under the control of the RCP application developer. A variety of

[18] http://sofa.ow2.org/

tools which do not fit to the IDE category can be built on RCP including GUI for GIS systems, or applications for stock trading.

The development process of the Eclipse plug-ins and RCP applications is similar. 'Same gears are under the hood of plug-ins and RCP applications.' The Plug-in based architecture of the Eclipse Platform with core technologies is used in both cases and generally it is easy to transform a plug-in into the RCP application or vice-versa.

5.1.2 GMF Ecore Diagram Editor

The Ecore Diagram Editor can be used for visualizing and editing of Ecore based EMF models and it provides an alternative to an original EMF tree-based editor for Ecore based models. User edits a model by drawing EMF *EClasses*[19] with UML-style class diagrams with *EAttributes*, *EOperions* and *Class EAnnotations* inside of nodes. The (containment) *EReferences* are visualized as (composition) associations found in UML class diagrams. The inheritance of EClasses is visualized in the same way as an inheritance relationship in UML. Other properties (e.g. Type of an EAttribute or multiplicity constraints) can be edited in properties view in the same way as in the original EMF based tree editor.

[19] Concepts of the EMF Framework are explained i.e. in [5, 20]

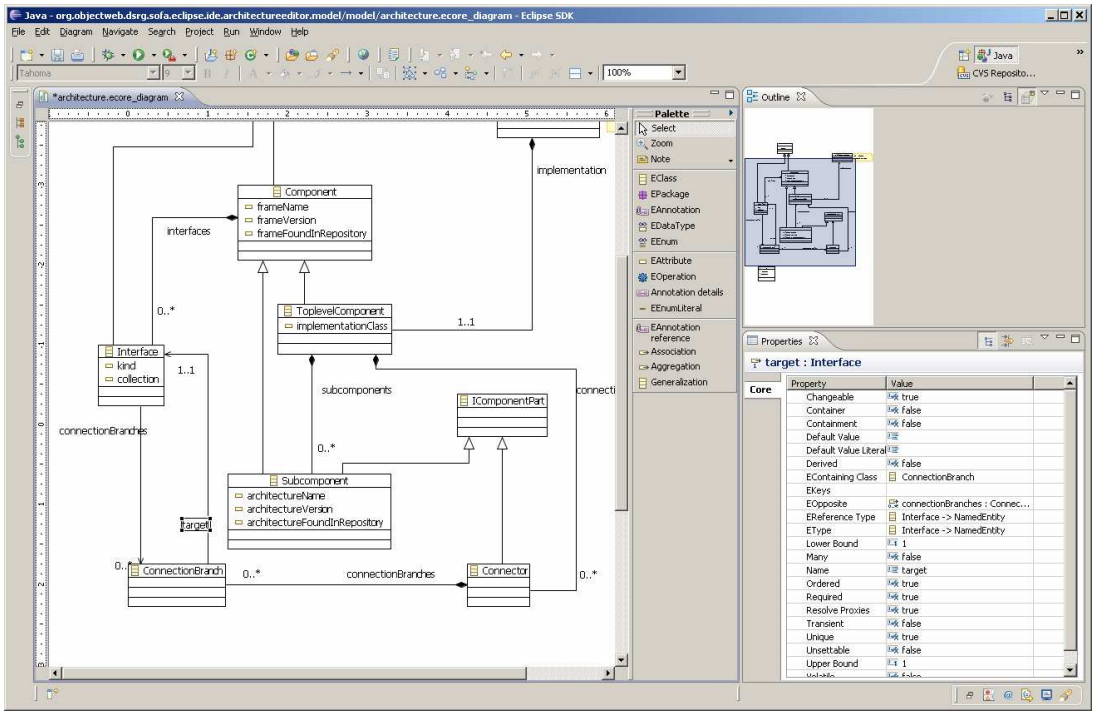


Figure 9 The Ecore Diagram Editor during the development of the (SOFA IDE) Architecture Diagram Model

The Ecore Diagram Editor is provided by authors of a GMF toolkit as an example to demonstrate a strength of a GMF Generator Model and according to authors it is fully defined within this GMF Generator Model without any customizations performed in generated Java classes. From the usability perspective of seeing it is probably its drawback as well. Small changes in the generated Java classes could probably lead in usability enhancements of this editor. Even so, a provided functionality makes this editor practical enough for a development of the Ecore based EMF models.

5.1.3 Spring IDE

The Spring IDE[20] is a GUI tool for creating of configuration files used in the *Spring Framework* project[21] (both open-source). It is created as a set of plug-ins

20 http://springide.org/

for the Eclipse Platform. Although the Spring Framework and SOFA 2.0 are two completely different systems and the relation between Spring IDE and Spring Framework is similar to the relation between SOFA IDE and SOFA 2.0 components system.

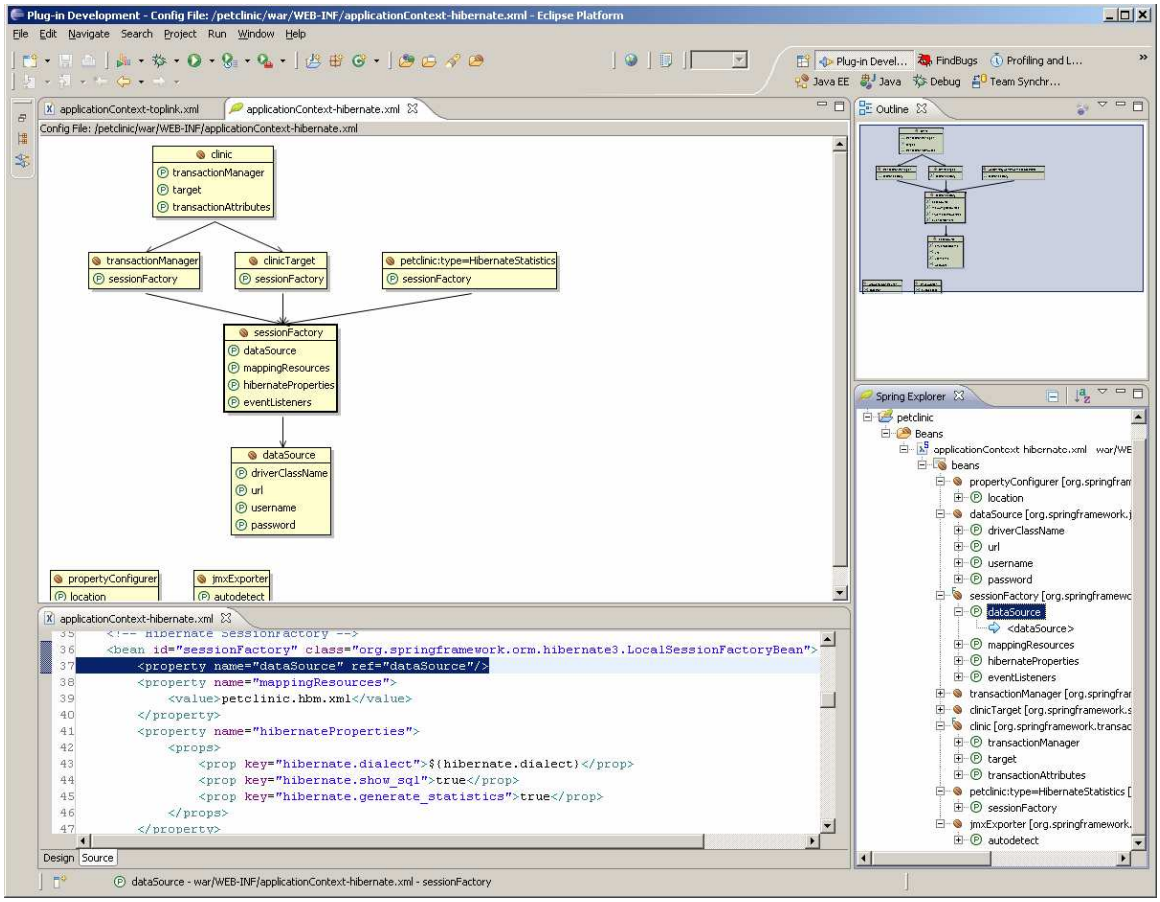


Figure 10 Spring IDE

The core of the *Spring IDE* consists from a *Beans Core Model* which is capable to resolve semantic information from *Spring Beans configuration files*. The information provided by this model is then reused in other parts of the Spring IDE which directly interact with a user.

A Beans Dependency Graph part provides a visualization of dependencies between *Spring Beans* by drawing their dependency graph. User

[21] http://www.springframework.org/

can efficiently use this feature to take himself in control by gaining a quick overview of the whole configuration. The resulting diagram is not directly editable. It can be just used to navigate to an appropriate context in XML configuration files by clicking on the nodes. The layout of the graph cannot be hand-edited but in this case this is irrelevant, because dependencies among *Spring Beans* have to form an acyclic graph. The hierarchical layout algorithms for drawing acyclic graphs are well studied in [3, 24]] and the automatic layout provided by *Spring IDE* achieves well-arranged and aesthetically appealing results.

For editing *Spring Beans* configuration files there is an enhancement to XML editor provided by Spring IDE and originated in the Eclipse WTP[22]. It implements a content assistant for auto-suggesting as user types. The assistant is based on semantic meaning of a given context which can be resolved from the *Beans Core Model* – if an attribute should contain a fully qualified name of the Java Class, the class-names matching a typed prefix are shown in an auto-suggest window. The same for bean IDs, property names, etc. Similar functionality is provided by the Eclipse Java editor or *IntelliSense* feature implemented in multiple Microsoft products for developers.

A Beans Project Validator is a part of Spring IDE providing a semantic validation of the Spring Beans configuration (e.g. if they do not refer to non-existing beans, non-instantiate abstract classes, etc.) This functionality also profits from the custom *Beans Core Model* and it is beyond capabilities of general XML tools, like an XML Schema validation.

There is no concept of a centralized *component repository* for *Spring Beans* compared to SOFA. Thus, there is no counterpart to the SOFA IDE Repository Browser able to connect multiple component repositories. The Spring Beans are XML file-based resources and user can simply mark which

[22] Eclipse Web Tools Platform Project – http://www.eclipse.org/webtools/

files in an Eclipse Project should be resolved by the Spring Core model and propagated to another parts of the Spring IDE.

6 Conclusion

This thesis has aimed on creation of prototype version of SOFA IDE. It allows visual development of applications for the SOFA 2.0 system [6] composed of hierarchical components and managing the component repository. SOFA IDE has been implemented as an Eclipse plug-in. It consists of four main parts: Repository Browser, SOFA Projects Manager, Repository Interaction Actions, and Architecture Diagram Editor.

The Repository Browser part allows exploring content of connected component repositories as well as retrieval of component elements to SOFA Project (using Repository Interaction Actions). The SOFA Projects Manager manages SOFA-centric information about component elements located in Resources of a SOFA Project within Eclipse workspace. Repository Interaction Actions implement communication with component repository needed for retrieval of component elements from a component repository to a SOFA Project, and for writing changed component elements back to the component repository. The Architecture Diagram Editor enables users to develop *hierarchical architectures* visually by using diagrams similar to UML 2.0 component diagrams.

The Architecture Diagram Editor part has been created using the Graphical Modeling Framework (GMF). We have found that GMF is a suitable tool for this purpose and can help shorten development time of a diagram editor. Generative approach used by GMF does not hide developer from internal complexities of the diagram editor implementation and there is “more

than one way" how to achieve a desired result. This is powerful because GMF does not limit developers in providing customizations to generated code. However, the learning curve of GMF is nontrivial.

During development of the SOFA IDE, we have found some bugs in the SOFA repository management API related to a new way how it is used from a "long-running" process of GUI application with multiple instances co-existing side-by side within one JVM.

The prototype implementation of SOFA IDE provides proof-of-concept for IDE tool for the SOFA 2.0 system with basic functionality. There is space for many enhancements to this initial solution. Examples are provided in the next section.

6.1 Future Work

There are many possibilities, how the SOFA IDE can be enhanced. Further analysis of user requirements based on the experience with usage of the prototype implementation is needed. These enhancements can include e.g.:

- Repository Search – "Find all *frames* with *interface* named 'ItfAbc' of *interface type* 'ItfTypeXyz'
- A GUI editor for *frames*
- Visual diff / merge support for synchronization with the repository and revealing changes between different versions of component elements
- Actions for Repository Cloning
- Drag & drop support for instantiating sub-components in the Architecture Diagram Editor from frames and architectures listed in the Repository Browser
- Auto-layout for the Architecture Diagram Editor

- Extend the Architecture Diagram Editor to support multiple hierarchies of nested components

- Validation of SOFA 2.0 component model can be integrated with the SOFA IDE and *Markers*[23] can be used to reveal context of found errors

SOFA 2 Management Console (see Section 5.1.1) is another tool for the SOFA 2.0 system. It is a complementary part to SOFA IDE. It provides management of the SOFA 2.0 runtime environment. The SOFA 2 Management Console is also developed by using Eclipse Platform. The SOFA IDE and the SOFA 2 Management Console can be easily integrated together to provide functionality for testing and debugging of component applications.

[23] Eclipse Platform uses *Markers* to reveal problems in source codes (e.g. compilation errors). Markers are presented in the **Problems** view in consolidated form and are also visually revealed as "exclamation marks" overlays on Resource icons in the *Projects Explorer,* or in the margin of a text editor.

Appendix A How to Retrieve SOFA IDE

Binary and source packages with prototype version of SOFA IDE plug-in can be retrieved from `http://urtax.ms.mff.cuni.cz/~pivoluska/sofaide/`

SOFA 2 distributed component system can be retrieved from project home-page (`http://sofa.ow2.org/`).

SOFA IDE plug-in requires at least Eclipse 3.3 with EMF and GMF plug-ins installed. To install SOFA IDE plug-in, simply extract the content of binary package with SOFA IDE plug-in to Eclipse installation directory and restart Eclipse. SOFA 2.0 API–related dependencies are bundled with plug-in. (Eclipse Update Site is not provided for prototype version of SOFA IDE.)

SOFA IDE plug-in has been developed and tested with SOFA 2.0 package v0.4.1.

Bibliography

[1] *GMF Programmer's Guide*.

[2] The Eclipse Modeling Framework (EMF) Overview.

[3] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[4] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[5] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[6] Tomáš Bureš, Petr Hnětynka, and František Plášil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of the International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Tomáš Bureš, Petr Hnětynka, and František Plášil. Runtime concepts of hierarchical software components. *International Journal of Computer & Information Science*, 8(S):454–463, 2007.

[8] Tomáš Bureš, Hnětynka Petr, František Plášil, Ján Klesnil, Kmoch Ondrej, Kohan Tomáš, and Pavel Kotrč. Runtime support for advanced component concepts. In *SERA '07: Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications*, pages 337–345, Washington, DC, USA, 2007. IEEE Computer Society.

[9] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Eclipse. Addison Wesley Professional, second edition, 2006.

[10] Distributed Systems Research Group, Department of Software Engineering, Faculty of Mathemathics and Physics, Charles University. *SOFA 2.0 Documentation*, 2007.

[11] Eclipse Foundation. *Eclipse Documentation*.

[12] The Eclipse Foundation. *Eclipse – User Interface Guidelines*, 2008.

[13] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[15] Petr Hnětynka and František Plášil. Distributed versioning model for MOF. In *WISICT '04: Proceedings of the winter international symposium on Information and communication technologies*, pages 1–6. Trinity College Dublin, 2004.

[16] Petr Hnětynka and František Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, editors, *CBSE*, volume 4063 of *Lecture Notes in Computer Science*, pages 352–359. Springer, 2006.

[17] Petr Hnětynka, František Plášil, Tomáš Bureš, Vladimír Mencl, and Lucia Kapová. Sofa 2.0 metamodel. Technical Report 2005/11, Department of Software Engineering, Faculty of Mathemathics and Physics, Charles University, 2005.

[18] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, July 2005.

[19] Doug McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[20] William Moore, David Bean, Anna Gerber, Gunnar Wagenknecht, and Phillipe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. IBM, first edition, 2004.

[21] Michael Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[22] Object Management Group. *Meta Object Facility (MOF) Core Specification v 2.0 OMG Available Specification formal/06-01-01*, 2006.

[23] Object Management Group. *XMI Specification, v 2.1.1, OMG document formal/2007-12-01*, 12 2007.

[24] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. SMC-11(2):109–125, February 1981.

[25] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[26] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[29] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.